Une image contenant texte, capture d’écran, Police, nombre

Description générée automatiquement

•Somme de termes - ignorer les termes croissant moins vite que les autres

•Produits de facteurs - ignorer les facteurs constants

•Ordre de grandeur

**ATTENTION :** Si les boucles ne sont pas imbriquées, on prend la complexité la plus élevée.

**NE PAS OUBLIER :** Des codes à la suite les complexités s’additionnent, mais les codes qui s’imbriquent, elles se multiplient.

|  |  |  |
| --- | --- | --- |
| **Fonctions de la STL** | | |
| **Fonctions** | **complexité** | **détails** |
| std::nth\_element |  |  |
| std::find\_XX |  |  |
| std::generate |  | n étant la taille du vecteur à remplir |
| std::distance |  |  |
| std::max/min\_element |  |  |
| std::unique |  |  |
| std::search |  |  |
| std::search\_n |  |  |
| std::swap |  |  |
| std::merge |  | M et N étant la taille des listes |
| std::sort |  |  |
| std::stable\_sort |  | si assez de mémoire |
| std::partial\_sort |  | où et |
| std::upper/lower\_bound |  |  |
| std::equal\_range |  |  |
| std::binary\_search |  |  |

Si on appelle x fois la fonction et que le paramètre diminue de 1 à chaque fois on  : f(n-1) + f(n-1) + f(n-1) et que n = 4 la réponse sera de donc

Un second exemple pourrait être d’appeler x fois la fonction et de diviser par 3 à chaque fois, dans ce cas

**Si on a des appels à (f-1) et (f-2), la complexité est équivalent à**  ou

= le nombre d’or.

Une image contenant texte, capture d’écran, Police, nombre

Description générée automatiquement

**Arbre des appels récursifs** **:** Affichage avant les 2 appels (1), affichage entre les deux appels (2) et affichage après les 2 appels (3).

Une image contenant texte, diagramme, ligne, Tracé

Description générée automatiquement

**Tri à bulles (Bubble sort) :** parcourt le tableau du début vers la fin et amène à la fin l’élément le plus grand. Il fait cela en boucle jusqu’à avoir parcouru la taille du tableau.

**Tri par sélection (Selection sort) :** Rechercher le plus petit élément du tableau, et l'échanger avec l'élément d'indice 0 puis rechercher le second plus petit élément du tableau, et l'échanger avec l'élément d'indice 1. Continuer de cette façon jusqu'à ce que le tableau soit entièrement trié.

Une image contenant texte, Police, blanc, outil

Description générée automatiquement

**Tri par insertion (Insertion sort)  :** Pour trouver la place où insérer un élément parmi les précédents, il faut le comparer à ces derniers, et les décaler afin de libérer une place où effectuer l'insertion. Le décalage occupe la place laissée libre par l'élément considéré. En pratique, ces deux actions s'effectuent en une passe, qui consiste à faire « remonter » l'élément au fur et à mesure jusqu'à rencontrer un élément plus petit.

Une image contenant Police, capture d’écran, texte, typographie

Description générée automatiquement

**Tri fusion (Merge sort) :** Algorithme récursif dans lequel on essaye de trier un tableau en le séparant en tableaux de 1 élément trié. Une fois les tableaux séparés, on les fusionne au fur et à mesure. Les petits tableaux sont dans l’ordre, il suffit de comparer à chaque fois le premier élément de chaque tableau et de le mettre dans le tableau de destination. La complexité est linéarithmique. Il n’est pas en place, mais il est stable. Pas adapter aux petits tableaux.

Une image contenant texte, diagramme, Plan, ligne

Description générée automatiquement

**Tri rapide (Quick sort) :** On choisit un pivot et on le met en fin de tableau, puis on met à gauche tous les éléments plus petits et à droite les plus grand. Il est stable et en place. Pour se faire, on a deux itérateurs qui s’arrêtent lorsque un élément est à la mauvaise place et les inverse. La manipulation s’arrête lorsqu’ils se croisent. **Si les valeurs sont égales on inverse !!!**

**Une image contenant texte, capture d’écran, nombre, Police

Description générée automatiquement**

**Tips détecté les tris :**

* **Tri à bulles :** Les plus grandes valeurs se retrouvent à la fin.
* **Tri par insertion :** Début organisé, mais la fin ne l’est pas encore (on peut voir des petites valeurs).
* **Tri par sélection :** Début organisé et il n’y pas de plus petites valeurs autre part qu’au début.
* **Tri par fusion :** On retrouve des morceaux triés dans certains partie des éléments. Séparer en deux à chaque fois.
* **Tri de Shell :** Le tri n’a aucun sens.
* **Tri rapide** **:** Il faut essayer de trouver quel était le pivot et selon quoi le tableau est plus ou moins mélanger.

**std::qsort :** En moyenne dans le pire des cas . Pas stable.

**std::sort :** , souvent une variation du **tri rapide**, pas stable.

**std::stable\_sort** : garantit une complexité temporelle en , si il y a assez de mémoire ou une complexité temporelle en si le tri fusion doit être réalisé en place. Il est stable, moins rapide que **sdt::sort**, les complexités sont garanties dans le pire des cas. Le **std::move(T)** doit être implémenté pour le type T car c’est une variante du **tri fusion**.

**std::nth\_element** **:** Prend l’élément les itérateurs d’un tableau et au milieu le pilier pour partitionner et fait une partition.

**std::partial\_sort :**  avec ***n = last - first*** et ***m = middle – first***

Une image contenant texte, capture d’écran, Police, nombre

Description générée automatiquement

Une image contenant texte, capture d’écran, Police, nombre

Description générée automatiquement

**Variables globales :** construites et allouées avant le main() et détruites et libérées après le main().

**Variables statiques :** construites lors du premier passage et détruites et libérées après le main().

**Variables automatiques :** construites lors de leur exécution et détruite lors de la sortie du bloc.

**Variables dynamiques :** construites au passage de l’exécution sur « new » et détruites et libérées au passage sur « delete ».

Une image contenant texte, Police, capture d’écran, ligne

Description générée automatiquementUne image contenant texte, Police, capture d’écran, ligne

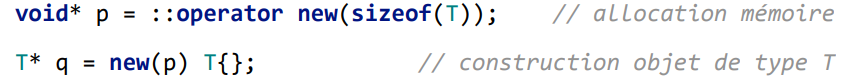
Description générée automatiquement

On détruit dans le sens inverse de la construction. Le premier élément créé est le dernier élément détruit.

Le mot clé « new » alloue et construit, mais il est possible de séparer les deux.



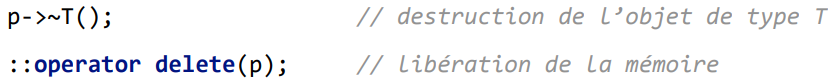
cette ligne devient alors les deux lignes ci-dessous



La même manipulation est possible lors de la destruction et libération avec « delete ».



cette ligne devient alors les deux lignes ci-dessous.



**Fuites mémoires :** Perdre l’adresse de l’objet à détruire, perdre le pointeur en sortant du scope ou une exception est levée. Pour les éviter, il faut encapsuler les allocations dynamiques et détruire dans les destructeurs.

Il faut implémenter **std::swap** si on souhaite faire des tris (tri à bulles en a besoin, par contre insertion et fusion non). C’est un simple échange des adresses mémoire.

Il faut aussi implémenter tout le reste des opérateurs. Comme l’opérateur d’affichage et de comparaison pour la réalisation d’un tri à bulles.

Il faut implémenter **std::move** si on souhaite réaliser un tri par insertion générique. Le but étant de déplacer les ressources de a à b, mais en laissant a dans un état valide. **std::move** cast les paramètres en rvalue référence (T&&). Il va donc voler les ressources de la source et les rendre valide.

std::push\_back appelle le constructeur std::move et std::emplace\_back non.

Si on std::push\_back et qu’il n’y a pas assez de place, la taille double et on recopie tous les éléments et on n’oublie pas de détruire derrière.

Une image contenant texte, reçu, capture d’écran, Police

Description générée automatiquementUne image contenant texte, Police, capture d’écran, blanc

Description générée automatiquement



Le mieux est d’utiliser std::swap pour tous les types simples de notre classe.

**Une image contenant texte, Police, capture d’écran, écriture manuscrite

Description générée automatiquementUne image contenant texte, Police, capture d’écran, écriture manuscrite

Description générée automatiquement**

**Une image contenant texte, Police, capture d’écran, écriture manuscrite

Description générée automatiquement**

**Rappels :**

* **Reserve :** Non-destructif, par contre ça n’agrandit pas le vecteur après coup.
* **Operator new/delete :** N’appelle pas les constructeurs/destructeurs
* **new T(params.) :** Appel au constructeur avec params.
* **Static :** Alloc. Comme les autres, par contre destruction à la fin du programme.
* **vector<T> v(N)** : Construit N objets de type T
* **push\_back** : Il réalloue de la mémoire s’il n’y a pas assez de place en doublant la capacité. Il appelle le constructeur de copie pour mettre les objets dans le nouveau vecteur. Il finit par détruire les anciens éléments. Si l’objet existe déjà et qu’on l’ajoute, il passe par le constructeur de copie.
* **push\_back**(T(…)) : Il créer un objet avec le constructeur normal, puis appelle le constructeur de déplacement et détruit le temp.
* **emplace\_back**(arguments de la classe) : Construit directement dans le vecteur. Pas de move, appelle constructeur basique. Juste un appel au constructeur normal (en place).
* **resize(n, val) :** 
  + n < taille : On détruit les éléments
  + n > taille : On crée un objet val. avec le constructeur et on fait des copies avec le constructeur de copie. Si pas de valeur précisée, alors on crée un objet avec le constructeur par défaut et on copie.
  + n > capacité : On realloue comme push\_back
* **swap**: Fait une construction avec le constructeur de déplacement et fait deux affectations avec l’opérateur d’affectation par déplacement. Puis détruit l’objet temporaire.
* **v[i] = T(…)** : Construit l’objet normalement et fait une affectation par déplacement, puis détruit.
* **V[i] = T**: Appel l’affectation par copie.
* Si on initialise un objet avec un **std::move**, ça appelle le constructeur de déplacement. Et si on fait une affectation avec, ça appelle l’affectation par déplacement. Un move permet d’éviter une destruction.

**Si on construit un objet à l’aide d’un objet temporaire, le constructeur ne construit qu’un seul objet.**

**A la fin d’un bloc on détruit tous les objets existants. Dans le cas d’un vecteur, on détruit tous les objets dans ce vecteur un à un.**

|  |  |  |
| --- | --- | --- |
| **Déclaration statique** | Alloc. comme les autres, par contre destruction à la fin du programme. Avec params ou défaut. | Cp ou Cd |
| **Reserve(M)** | Non-destructif. Appel du constructeur de déplacement (Cm) et destructeur (D), N fois. Par exemple s’il y avait 3 objets dans le vecteur, on aurait CmCmCmDDD. **!! Alloue de la taille au vecteur pas de doulage !!** | M \* (Cm D) |
| **new T(params./void)** | Appel au constructeur avec params | Cp ou Cd |
| **Operator new/delete** | N’appelle pas les constructeurs/destructeurs | - |
| **vector<T> v(N)** | Construit M objets de type T. Donc utilise le constructeur par défaut. | M \* Cd |
| **push\_back** | Il réalloue de la mémoire s’il n’y a pas assez de place en doublant la capacité. Il appelle le constructeur de copie pour mettre les objets dans le nouveau vecteur. Il finit par détruire les anciens éléments. Si l’objet existe déjà et qu’on l’ajoute, il passe par le constructeur de copie. |  |
| **push\_back**(T(…)) | Il créer un objet avec le constructeur normal, puis appelle le constructeur de déplacement et détruit le temp. | CpCmD |
| **emplace\_back(T(…))** | Construit directement dans le vecteur. Pas de move, appelle constructeur basique. Juste un appel au constructeur normal (en place). | Cp/Cd |
| **resize(M, val)** | M < taille : On détruit les éléments  M > taille : On crée un objet val. avec le constructeur et on fait des copies avec le constructeur de copie. Si pas de valeur précisée, alors on crée un objet avec le constructeur par défaut et on copie.  M > capacité : On réalloue comme push\_back |  |
| **swap** | Fait une construction avec le constructeur de déplacement et fait deux affectations avec l’opérateur d’affectation par déplacement. Puis détruit l’objet temporaire. | CmAmAmD |
| **v[i] = T(…)** | Construit l’objet normalement et fait une affectation par déplacement, puis détruit. | CpAmD |
| **V[i] = T** | Affecter un objet qui existe déjà. Appel l’affectation par copie. | Ac |
| **std::move** | Si on initialise un objet avec un std::move, ça appelle le constructeur de déplacement. Et si on fait une affectation avec, ça appelle l’affectation par déplacement. Un move permet d’éviter une destruction | Cm ou Am |
| **Erase(begin())** | Déplace N-1 objet vers la gauche avec affectation par déplacement et détruit 1 fois | (N-1)\*Am D |
| **T\* toto** | Ne fait rien, car ça déclare un pointeur et non un objet. |  |
| **Destruction** | En sortie des accolades ou en fin de programme. Sauf allocation dynamique et static qui doit être free. | D |

* **Si on construit un objet à l’aide d’un objet temporaire, le constructeur ne construit qu’un seul objet.**
* **A la fin d’un bloc on détruit tous les objets existants. Dans le cas d’un vecteur, on détruit tous les objets dans ce vecteur un à un.**

Une image contenant table

Description générée automatiquement

**Tableaux de taille fixe (std::array<T, N>) :** Pas d’insertion / suppression d’éléments, Accès aux éléments en O(1) en calculant leur adresse.

Une image contenant ligne, diagramme, Police, capture d’écran

Description générée automatiquementUne image contenant texte, Police, capture d’écran, blanc

Description générée automatiquement

**Buffer circulaire :** Tableau de taille variable indexé de 0 à taille-1 efficace pour insérer / supprimer en fin pas efficace pour insérer / supprimer ailleurs qu’en fin. File FIFO, Le buffer circulaire remplace l’index 0 par un index début.



Avantages : pour insérer/supprimer en début ou en fin , accéder en toute position

Inconvénients : Insérer ou supprimer à l’indice i en , capacité fixe.

**Tableaux de capacité variable (std::vector<T>)  :**

Une image contenant texte, capture d’écran, Police, nombre

Description générée automatiquement

**Listes simplement chainées (std::forward\_list<T>) :** Tableaux inefficaces pour insérer / supprimer en positions autres que fin ( et début ). Ne pas stocker les éléments dans des emplacements mémoire consécutifs, on perd la relation suivant/précédent implicite. Chaque élément est stocké dans une structure appelée maillon qui inclut un pointeur vers le maillon suivant ( et précédent ) L’ensemble de ces maillons constitue une liste chainée.

Une image contenant ligne, diagramme, capture d’écran, Rectangle

Description générée automatiquement Une image contenant texte, Police, blanc, capture d’écran

Description générée automatiquement

Une image contenant texte, capture d’écran, Police, nombre

Description générée automatiquement

**Listes doublements chainées (std::list<T>) :** Comme la liste simple, sauf qu’il y a aussi un pointeur vers le maillon précédent.Une image contenant Police, Graphique, ligne, conception

Description générée automatiquement Une image contenant texte, Police, capture d’écran, blanc

Description générée automatiquement

Une image contenant texte, Police, blanc, capture d’écran

Description générée automatiquement

Une image contenant texte, capture d’écran, Police, nombre

Description générée automatiquement

Double ended queue (std::deque<T>) : Opérations en début et en fin en temps amorti constant, accès indexé en temps constant

Une image contenant texte, capture d’écran, diagramme, Police

Description générée automatiquementUne image contenant texte, capture d’écran, Police, nombre

Description générée automatiquement

Une image contenant texte, capture d’écran, nombre, Police

Description générée automatiquement

**std::list<T>**

Une image contenant texte, capture d’écran, nombre, Police

Description générée automatiquement

Une image contenant texte, capture d’écran, Police, nombre

Description générée automatiquement

**std::forward\_list<T>**

Une image contenant texte, capture d’écran, Police, nombre

Description générée automatiquement

**std::deque<T>**

**Une image contenant texte, capture d’écran, Police, ligne

Description générée automatiquement**

**Un tas (heap) :** est un arbre binaire complet dont tous les éléments sont plus petits ou égaux que leurs parents, c’est la condition de tas (CDT).

**push\_back :** Ajoute la valeur en fin de vecteur, mais pas dans le tas.

**pop\_back :** Supprime le dernier élément du vecteur.

**Insertion (push\_heap) :** On insère toujours un élément à la fin du tas et on le remonte après. L’insertion est en et la remontée est en .

Une image contenant cercle, diagramme, illustration

Description générée automatiquement

**Supprimer le sommet du tas (pop\_heap) :** Il faut swap le sommet avec le dernier élément et réorganiser le tas. Pour réorganiser, il faut faire descendre le sommet (celui que l’on vient d’échanger) en direction du plus grand des enfants. **Attention :** l’élément supprimer reste dans le tableau qui stocke le tas.

Une image contenant cercle, diagramme, capture d’écran, conception

Description générée automatiquementUne image contenant cercle, clipart, dessin humoristique, diagramme

Description générée automatiquement

**Créer un tas (make\_heap) :** On insère les éléments en fin et ensuite on les fait remonter à leur place. Ici, on insère les éléments dans l’ordre donc du deuxième au dernier. C’est un algorithme en Remonter éléments.

Une image contenant capture d’écran, diagramme, ligne

Description générée automatiquement

On fait les nœuds du dernier au premier, mais on prend en compte que les nœuds qui ont des enfants et si la CDT n’est pas respectée pour le nœud, alors on le fait descendre le nœud jusqu’à sa place en swappant avec le plus grand enfant. Complexité en O(N). Descendre éléments.

Une image contenant capture d’écran, diagramme

Description générée automatiquement

**Tri par tas (sort\_heap) :** Il faut déjà avoir un tas, puis on swap le premier et dernier élément et on ré-organise la condition de tas en descendant le premier élément. Sélection maximum en et tri en

**!! POUR LE TRI, IL FAUT QUE LA CDT SOIT RESPECTER SINON MARCHE PAS !!**

Dans la STL, il existe des fonctions pour faire ces algorithmes (make\_heap, sort, pop, push, is). Elles ne vérifient pas que c’est bien un tas, c’est à nous de vérifier (pop et sort 🡪 tas entre [first ; last( et push 🡪 tas entre [first ; last-1(.

|  |  |
| --- | --- |
| **Make\_heap** | 3\*std::distance(first, last) |
| **Sort\_heap** |  |
| **Push \_heap** |  |
| **Pop\_heap** |  |
| **Is\_heap(\_until)** |  |

**!! Si le tas respecte déjà la CDT, insérer un élément se fait en O(1) !!**

**!! Ces structures ont la même complexité qu’un tas !!**

**Pile (std::stack<T, Container>) :** structure où l’on insère et supprime du même côté (au sommet). LIFO. Elle permet notamment d’évaluer des expressions. NPI c’est lorsque l’on met l’opérateur à la fin de l’expression (permet d’éliminer les parenthèses).

Une image contenant capture d’écran, texte, ligne, Police

Description générée automatiquement

Une image contenant texte, capture d’écran, Parallèle, nombre

Description générée automatiquement

**Notations polonaise inverse :** Pour évaluer, l’ordre opérateur / opérande n’importe pas.

Une image contenant Police, texte, nombre, capture d’écran

Description générée automatiquement

En plaçant opérateur après ses opérandes, plus besoin de parenthèses



**File (std::queue <T, Container>) :** On insère d’un côté et on supprime de l’autre, FIFO. On peut donc accéder aussi accéder au dernier élément contrairement à LIFO.

Une image contenant texte, capture d’écran, ligne, Police

Description générée automatiquement

**File de priorité (std::priority\_queue<T, Container, Compare>) :** File organisée selon un critère de priorité, mise en œuvre avec un tas. On peut toujours accéder à l’élément le plus prioritaire. Le sommet de la file est l’élément le plus prioritaire. Le sommet est donc l’élément ayant le plus grand indice. Si il y a une égalité, on compare les valeurs.

|  |  |
| --- | --- |
| Top (front) | Accède au sommet (+ grand/prior.) |
| Back | Accède au dernier élément (file) |
| Push | Insérer un élément |
| Pop | Supprime le sommet |

Racine : N’a pas de parent.

Nœuds internes : Nœuds qui ont un ou plusieurs enfants.

Feuilles : N’ont pas d’enfants.

**Degré nœud :** Nombre d’enfants. / Degré arbre : Degré max de ses nœuds.

**Chemin d’un nœud :** Suite des nœuds reliant la racine au nœud.

**Niveau d’un nœud :** Longueur de son chemin (Ex : L 🡪 4).

**Hauteur d’un arbre :** Niveau maximum parmi ses nœuds.

Une image contenant diagramme, cercle

Description générée automatiquement

**Vide :** Sans aucun nœud et a une hauteur de 0.

**Plein :** Tous les nœuds de niveau inférieur à h-1 sont de degré d. Les nœuds de niveau h-1 ont un degré quelconque. Les nœuds de niveau h sont des feuilles.

**Complet :** Arbre plein dont le dernier niveau est rempli par la gauche.

**Dégénéré :** Arbre de degré 1. Type liste chaînée.

**Binaire :** Arbre de degré <= 2. On distingue enfant gauche et droite.

Une image contenant clipart, Dessin d’enfant, dessin humoristique, illustration

Description générée automatiquement

**Attention :** Il ne faut pas oublier qu’un nœud a un pointeur sur ses enfants, son parent et son puiné (enfant droite du même parent). Parcours symétrique que pour binaire.

**Arbre à partir de deux parcours :** On utilise le parcours post ou pré pour connaître la racine de l’arbre et les racines des sous-arbres. Et on utilise le parcours symétrique pour savoir quel sommet est à gauche ou à droite du quel autre sommet. On va donc regarder le début du pré ordre pour trouver la racine. Ensuit le symétrique pour trouver les sous arbres gauche et droite. Après on regarde le pré ordre de gauche à droite pour savoir quel est le sommet qui vient en premier dans le sous-arbre gauche, etc…

**Expressions arithmétiques :** Arbre binaire avec 2 types de nœuds, feuille = valeurs et nœuds interne = opérateurs binaires tels que +, -, \*, /. Pas de nœud de degré 1.

* Infixe : (((1+2)/(3-4))\*(1-2))
* Préfixe : \*/+12-34-12
* Postfixe : 12+34-/12-\*

**Arbre binaire de recherche (ABR) :** Arbre binaire où l’enfant gauche est toujours plus petit que son parent et l’enfant droite est toujours plus grand que son parent. Pour chercher/insérer/supprimer une clé, on retrouve une complexité en .

**ABR chercher/insérer :** il faut comparer la clé avec la racine et descendre en fonction de si la clé est plus petite (à gauche) ou plus grande (à droite).

**ABR supprimer :** Si on supprime des :

* **Feuilles :** il ne se passe rien,
* **Nœuds de degré 1 :** il suffit de raccrocher le sous-arbre.
* **Nœuds de degré 2 ou plus :** utiliser la technique de Hibbard.

**Technique de Hibbard :** on prend le minimum du sous-arbre droit ou le maximum du sous-arbre gauche.

Une image contenant dessin humoristique, croquis, illustration, art

Description générée automatiquement

**Equilibre :** Un arbre est équilibré si

Une image contenant texte, capture d’écran, Police, ligne

Description générée automatiquementUne image contenant capture d’écran, cercle, diagramme, conception

Description générée automatiquement

**Arbre AVL (AVL tree) :** Un arbre AVL est un arbre binaire de recherche qui garantit qu'aucun nœud n'a un déséquilibre autre que -1, 0 ou 1 en se rééquilibrant à chaque opération d’insertion / suppression.

**AVL insérer :** même que pour ABR avec un plus un rééquilibrage après insertion.

**AVL supprimer :** même que pour ABR avec aussi un rééquilibrage après suppression.

**Équilibrage AVL :** l’équilibrage AVL est composé de 3 cas.

* Cas 1 : déséquilibre de -1, cas résous avec une rotation simple. rg(x).

Une image contenant diagramme, cercle, clipart, conception

Description générée automatiquement

* Cas 2 : déséquilibre de 0, cas résous avec une rotation simple. rg(x).

Une image contenant clipart, diagramme, cercle, dessin humoristique

Description générée automatiquement

* Cas 3 : déséquilibre de +1, il faut réaliser une double rotation. On effectue une rotation dans le sens inverse du nœud du bas et ensuite on effectue la rotation dans l’autre sens du nœud du dessus.

Une image contenant texte, dessin humoristique, diagramme, clipart

Description générée automatiquement

Niveau(hauteur) 🡪 Meilleur :  ; Pire 🡪

**TDA** : type de données abstrait

**TDA Ensemble :** TDA qui stocke les données sans répétition. Un ensemble est dit ordonné si on peut le parcourir dans l’ordre. C’est le cas quand il est mis en œuvre avec un ABR.

**TDA Tableau associatif :** Aussi appelé dictionnaire, Associe des clés uniques et des valeurs. Ce sont des std::map où on donne un type à la clé et un type à la valeur à stocker. C’est faisable avec des ABR en stockant clé et valeur. .

**Ensemble trié et unique (std::set<T, Compare>)****:** 3 pointeurs par nœud.

**End :** est un nœud au-dessus et il est vide.

**Begin :** est le nœud le plus à gauche

**Insertion :** par clé en , par indice (itérateur) en , plage de N clés dans un set de taille S en . Insertion d’un grand nombre de valeurs, mais les valeurs ont un petit domaine de définition se fait en .

**Suppression :** par clé et indice comme insertion, supprimer plage de valeur en .

**Complexités :**

Une image contenant texte, capture d’écran, Police, nombre

Description générée automatiquement

**Tableau associatif (std::map<Key, Value, Compare>) :** Comme un set, sauf qu’il contient des éléments de type « pair ». Il faut aussi une fonction de comparaison qui tri les clés. On accède à l’indice avec **.first** et à la valeur avec **.second**. Il fournit aussi l’opérateur[]. Il faut faire attention avec celui-ci, car une lecture d’une **valeur qui n’est pas présente l’ajoute dans la map** (utiliser find).

**Une image contenant texte, Police, capture d’écran, ligne

Description générée automatiquement**

**Complexités :**

**Une image contenant texte, capture d’écran, Police, nombre

Description générée automatiquement**

**Multiset/multimap :** Comme set et map, mais permet de stocker plusieurs fois la même clé. Pas d’opérateur [] pour les multimap et equal\_range retourne toutes les clés équivalentes.

**STL :** propose déjà les opérations ensemblistes. Pour ces opération (union, intersection), il faut passer les deux plages sur lesquels faire les opérations et la plage stockant le résultat.

**Consommation mémoire :**

|  |  |
| --- | --- |
| std::array | Aucun coût à part la taille des éléments |
| std::vector | 3 pointeurs (taille à vide d’un vector), éléments |
| std::forward\_list | 1 pointeur sur la liste, 1 pointeurs par élément + éléments |
| std::list | 2 pointeurs + 1 size\_t (taille à vide), 2 pointeurs par élément + éléments |
| std::deque | 6 pointeurs + n/c pointeurs + c éléments |
| std::set | 2 pointeurs + 1 size\_t (à vide), 1 maillon du set stocke 3 pointeurs et l’élément |
| std::map | 2 pointeurs + 1 size\_t (à vide), (3 pointeurs + clé) par élément + élément |

**Graphes non orientés :** Un graphe non orienté est formé, d’un ensemble dont les éléments sont appelés **sommets**. D’un ensemble dont les éléments sont appelés **arêtes**. D’une fonction d’incidence qui associe à chaque arête une paire de sommets appelés extrémités de l’arête .

Une image contenant cercle, ligne, diagramme

Description générée automatiquement

Une arête relie ses deux extrémités et et est **incidente** avec elles.

Deux sommets et sont dits **adjacents** s’il existe une arête qui les relie. Ils sont incidents à l’arête .

**Graphes orientés :** Un graphe orienté est formé, d’un ensemble dont les éléments sont appelés **sommets**. D’un ensemble dont les éléments sont appelés **arcs**. D’une fonction d’incidence qui associe à chaque arête une paire de sommets appelés extrémités de l’arête .

Une image contenant cercle, haltère

Description générée automatiquement

On appelle **l’extrémité initiale** et **l’extrémité finale** de l’arc .

On remplaçant tous les arcs de G par des arêtes de même extrémités, on obtient son graphe sous-jacent (non orienté).

**Parcours en largeur (breadth first search ou BFS) :** utilisation d’une file FIFO qui parcourt d’abord tous les sommets adjacents avant ceux à distance 2, puis 3, … BFS parcourt les sommets par ordre de distance croissante au sommet de départ.

Une image contenant diagramme, ligne, cercle

Description générée automatiquement

**Parcours en profondeurs (depth first search ou DFS) :** méthode récursive qui s’appelle pour tous les sommets adjacents, Le parcours depuis un sommet n’atteint pas nécessairement tous les sommets du graphe.

Une image contenant dessin, clipart, Dessin d’enfant, cercle

Description générée automatiquement

**Pré-ordre :** 5-2-0-1-3-4

**Post-ordre :** 1-0-4-3-2-5

**Complexités (N sommets et M liens) :**

|  |  |
| --- | --- |
| Liste des arêtes incidents à V (matrice) |  |
| Liste des arêtes incidents à V (liste) |  |
| Au total un parcours pour une matrice |  |
| Au total un parcours pour une liste |  |

**Dijkstra :** On stocke dans chaque sommet 2 informations. La distance (distTo(V)) jusqu’au sommet de départ et le dernier arc (edgeTo(V)) du chemin le plus court.On traite les sommets dans l’ordre de priorité de leur poids. Le poids le plus faible est celui à privilégier. Les poids sur le chemin le plus court final, sont les poids initiaux et pas cumulés.

**Complexités (N sommets et M arcs) :**

|  |  |
| --- | --- |
| Accès arcs sortant |  |
| Modifier la priorité d’un sommet (réalisé au pire à chaque relâchement de sommet donc M fois) |  |
| Complexité globale de Dijkstra |  |

**Tri topologique (topological sort) :** redessiner un graphe orienté acyclique (Directed Acyclic Graph ou DAG) pour que tous ses arcs pointent dans la même direction, Ce n’est évidemment pas possible pour un graphe comprenant un circuit. **Tri topologique = Inverse du post-ordre.**

**Détection de circuit :** Si le parcours nous amène à atteindre un sommet présent dans cette pile, tous les éléments de la pile entre ce sommet (vertex) et le sommet (top) de la pile forment un circuit.

**Algorithme de Kosaraju-Sharir :** Calculer le post ordre inverse du parcours en profondeur pour le graphe inverse de G

Une image contenant cercle, ligne, dessin, croquis

Description générée automatiquement

Pré-ordre : 0 2 3 4 5 11 9 6 7 8 12 10 1

Post-ordre : 5 7 8 6 10 12 9 11 4 3 2 0 1

Post inverse : 1 0 2 3 4 11 9 12 10 6 8 7 5

Finalement Calculer les composantes connexes par parcours en profondeur DFS sur G dans cet ordre :

Une image contenant dessin, Dessin d’enfant, clipart, illustration

Description générée automatiquement

DFS(1)->{ 1 }

DFS(0)->{ 0,5,4,2,3 }

DFS(11)->{ 11,12,9,10 }

DFS(6)->{ 6,8 }

DFS(7)->{ 7 }

**Étape :**

1. Trouver le DFS en post ordre et l’inverser.
2. Inverser les flèches sur le graphe.
3. Refaire des DFS avec le post ordre inverse pour trouver les composant connexes.

**Tables de Hachage :**

**Résolution de collisions par chainage :** Utilise un tableau de listes simplement chaînées pour stocker paires clé/valeur, avec . Transforme la clé en un entier

Une image contenant texte, Police, capture d’écran, diagramme

Description générée automatiquement

**Résolution de collisions par sondage linéaire :** insérer une clé si l’emplacement à l’indice est occupé, essayer , , … jusqu’à trouver un emplacement vide.

Chercher la clé si l’emplacement à l’indice est occupé mais ne correspond pas à ,essayer , , … jusqu’à trouver soit un emplacement vide.